Chapter 5

Scheduling – A Rectangle Packing Approach

In the last two chapters, we have seen test scheduling approaches based on TAM partitioning. Huang et al [79] first suggested a different approach in which the TAM lines are not physically partitioned. A test wire used as a part of test lines delivering patterns to a core-under-test, can become associated with a different set of wires while testing another one. This formulation gives rise to a bin-packing approach for test scheduling. The situation has been shown in Fig 5.1, in which TAM wires are assigned to four test sets over time. For a certain period of time, each test is assigned to some TAM wires. The problem that we concentrate on is, how to assign a start time, an end time and the set of TAM wires for each test in such a way that total test time is minimized. Another important issue in testing that has come up recently is about test power minimization. This is required as most of the chips today come up with a power budget. Thus, excessive power dissipation during test and the associated heat generated may cause permanent damage to the chip. Various strategies have been proposed in literature to reduce the test power. Based on these observations and due to the NP-hard nature of the TAM design algorithm, we have used a genetic algorithm (GA) based approach to solve the SOC test scheduling problem.

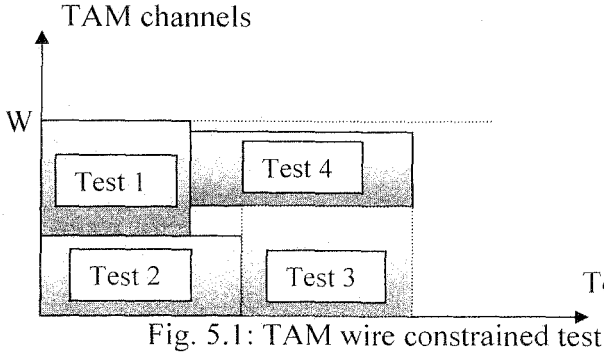


Fig. 5.1: TAM wire constrained test scheduling

The cores in an SOC are normally fitted with wrappers. The wrapper for a core facilitates the test application process. It can isolate the module from its surroundings and provide switching functionality between functional access to the modules and test access through the TAM. Though some cores may be equipped with wrappers, while others may not, in this work we consider only the wrapped cores. It is to be noted that the test time of a core depends on the length of the wrapper chains. The scan chains and the wrapper cells are configured to make a wrapper chain. An increasing number of wrapper chains reduce the test time due to the reduction in length of the wrapper chain. However, it takes more TAM wires and vice versa. We can formally describe the problem as follows.

Let the SOC design consist of N cores, and each core C_i $(1 \leq i \leq N)$ has

- n_i input terminals,
- m_i output terminals,
- bi-directional I/Os,
- S_i scan chains and
- for each scan chain k, the length of the scan chain (number of flip-flops) $l_{i,k}$.

Also assume that maximum peak power for each core during testing is given. Let, the total width of TAMs be W and each core must be tested with P_i patterns. So, the overall problem that we have to solve is as follows.

Given a set of N cores, their specific test parameters, the number of I/O pins for an SOC, maximum allowable peak power dissipation POW and peak power dissipation for each core, design the test schedule along with wrapper designs for all wrapper-based cores such that overall testing time is minimized and the peak power during testing never exceeds POW.

There are basically two steps in our approach to solve the problem. First, we generate possible optimized wrapper configurations for each core under specified TAM width. In the next step, we solve the test scheduling problem using the sets of optimized wrapper solutions under the maximum-allowable TAM width and power constraint. In our work, we consider the hard cores, for which, the number and length of the module-internal scan chains are fixed and cannot be changed any more while designing the SOC-level test architecture.

5.1 Core Wrapper Design

A test wrapper is the interface between the TAM and the core. Since larger cores typically have hundreds of terminals and the total number of TAM channels available depends on the limited number of SOC pins, wrappers facilitate test width adaptation when the TAM width is not equal to the number of core terminals. To design the wrapper for cores with internal scan chains, we have used the *Design_wrapper* algorithm proposed in [50]. To calculate test time T, for a wrapper we have used Eqn. 3.1.

Using the wrapper design method, for each core we can generate a set of wrapper configurations with the TAM wire usage of 1 to W, where W is the maximum number of TAM channels allocated to test the SOC. Hence, each wrapper configuration can be considered as a rectangle with width equal to the test time and height corresponding to the number of TAM wires allocated. So, each configuration is represented by a tuple $(w_{ij}, T(w_{ij}))$ - where, core i has been assigned a TAM width w_{ij} resulting in a test time $T(w_{ij})$ $(1 \leq j \leq W)$. From all the wrapper configurations for a core i, a smaller set of wrapper configurations need to be considered in the test scheduling. It is based on pareto-optimal design principle, where for a range of TAM widths, test time remains unchanged. Obviously, only pareto-optimal points are of interest since they make use of the lowest possible number of TAM channels to reach a certain test time. Table 5.1 shows the rectangles created for core 2 of SOC p93791.

TAM width	Testing time		TAM width	Testing time
1	7906		9	1155
2	4049		10-11	963
3	2891		12-13	962
4	2120		14-17	770
5	1734		18-19	769
6	1541		20-39	577
7	1348		40-64	385
8	1156			

Table 5.1: Rectangles created for core 2 of SOC p93791

5.2 Test Scheduling Problem

Suppose an SOC with N cores is to be tested using W TAM wires. Each core C_i $(1 \leq i \leq N)$ is represented by a set of wrapper configurations R_i. Each wrapper configuration is represented by a pair $(w_{ij}, T(w_{ij}))$, where w_{ij} stands for the width of the j-th wrapper configuration for core C_i and $T(w_{ij})$ is the test time of core C_i with wrapper width w_{ij}. Also, for each core, peak power during testing POW_i, is assumed to be available. So the objective is the assignment of core wrapper pins to the pins of SOC and for getting the test starting time and finishing time for each core such that overall test time is minimized satisfying power constraint.

This problem can be transformed into the well-known rectangle-packing problem, in which the SOC is represented by bin of width W and a set of R_i SOC wrappers for each core represented by a set of R_i rectangles with rectangle j having height w_{ij} and width $T(w_{ij})$. We want to choose one rectangle from each set of rectangles R_i and pack all the rectangles in the bin, so that width of the bin is minimized.

In this work, we treat this test scheduling problem as a collection of two different problems. First, we consider the test scheduling problem where no power constraint is considered. In the second one we impose the power constraint on test scheduling algorithm for the first problem and it is verified that for any time instant maximum allowable peak power POW (SOC power budget) will not be exceeded. To evaluate power requirement, we calculate the sum of the maximum peak power of all the cores under test at a given time instant. It is assumed that for entire test time of the core, the maximum peak power is same. So, it is needed to calculate only the sum of the peak powers when the testing of a core starts. Only one benchmark (h953) among the ITC'02 benchmark set has power dissipation numbers included. For other SOCs we have used the power consumption values for the tests in design p22810 and p93791 reported in Pouget et al [87] and given in Table 5.2.

5.3 Genetic Algorithm Based Approach

We now describe a genetic algorithm (GA) [88] based approach to solve the generalized version of rectangle packing that does not consider the power constraint of the SOC. The overview of the generalized rectangle problem using GA approach is as follows. Let us assume that W is the TAM width and N is the number of cores available. First of all, we calculate testing times for all cores for all possible widths, that is, *creating possible rectangles for all cores.* It is obvious that in the final solution, we have to select one rectangle form each set of rectangles available for all cores. The GA assigns precedence to cores to establish an order in which the cores are to be scheduled. We pack all the rectangles into the rectangular bin of height W using that order. A chromosome represents a solution identifying the rectangle to be used for a particular core and also the preference assigned to it.

Core	d695	p22810	p93791
1	660	173	7014
2	602	173	74
3	823	1238	69
4	275	80	225
5	690	64	248
6	354	112	6150
7	530	2489	41
8	753	144	41
9	641	148	77
10	1144	52	395
11	-	2505	862
12	-	289	4634
13	-	739	9741
14	-	848	9741
15	-	487	78
16	-	115	201
17	-	580	6674
18	-	237	113
19	-	442	5252
20	-	441	7670
21	-	167	113
22	-	318	76
23	-	1309	7844
24	-	260	21
25	-	363	45
26	-	311	76
27	-	2512	3135
28	-	2921	159
29	-	413	6756
30	-	598	77
31	-	-	218
32	-	-	396

Table 5.2: Power consumption values [87]

5.3.1 Solution representation

In the GA, we need to encode each solution as a chromosome. A chromosome in our approach consists of two parts namely *ordering part* and *rectangle-selection part.*

Ordering part of a chromosome gives the preference of the cores to be selected, *i.e.*, the order in which we select the cores one by one to pack into the rectangular bin. A core in the order will be packed into the rectangular bin if the available height is sufficient for the chosen core. Otherwise, the next core in that order will be chosen to be packed into the bin.

Rectangle-selection part of a chromosome tells us which rectangle to be selected from a set of rectangles for a core. This part randomly selects particular rectangles to be used for a solution for all cores. The number of rectangles available for a particular core is equal to the total TAM width. That is, if the TAM width or the rectangular bin height is equal to 32 then the number of rectangles available for a core is 32, each one corresponding to a varying width ranging from 1 to 32. It may be noted that we need not keep all these rectangles in the database – remembering only the pareto-optimal points suffices.

These two parts of the chromosome form a solution to the given problem.

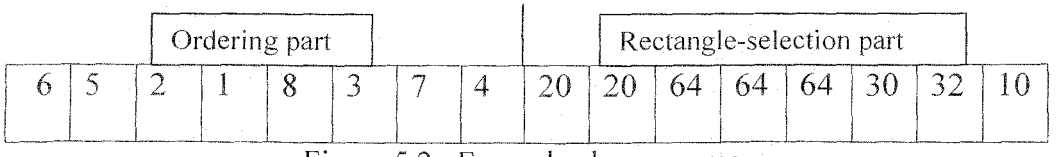


Figure 5.2: Example chromosome

Figure 5.2 shows an example chromosome. Here the rectangular bin width is 64 and the number of cores is 8. The ordering assigns sixth preference for core 1, fifth preference for core 2, second preference for core 3, first preference for core 4, and so on. The rectangle-selection part identifies the TAM widths for cores as follows: 20 for core1 and core2, 64 for cores 3, 4, and 5; 30 for

core 6, 32 for core 7, and 10 for core 8. This width assignment identifies the corresponding rectangle to be used for the core. The number of rectangles available for a core is assumed to be 64. Each rectangle gives different test application time depending on its height and test parameters. So, while packing the rectangles, we first consider core 4. Corresponding rectangle part gives the rectangle to be considered. In this case it is 64. Thus, the rectangle is selected and core 4 is scheduled from the beginning of test time for time units equal to the corresponding test time. At some point of time, let the bin be filled as shown in Fig 5.3. If the next core to be considered (depicted by chromosome) is core i, and the corresponding rectangle is r_i, we first check if r_i fits at start position t_1. If not, we try with the next core in the preference order. If for none of the cores the required rectangle fits starting from t_1, then the time t_1 to t_2 is left idle and we try to pack starting at t_2 again for core i. The same process is repeated.

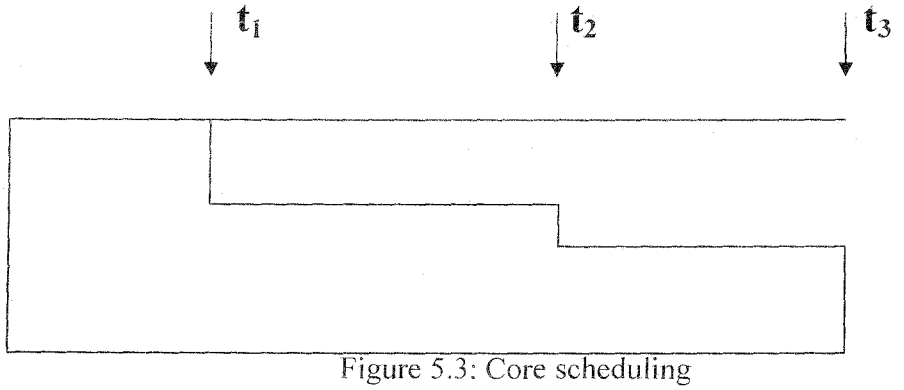


Figure 5.3: Core scheduling

5.3.2 Genetic operators

Two genetic operators crossover and mutation have been used to evolve new generations.

5.3.2.1 Crossover

Two chromosomes are selected randomly from the entire population. After selecting two chromosomes to participate in crossover, a single point

crossover is applied on the *ordering part* and *rectangle-selection part* of the chromosome. After generating new chromosomes, a check is made with the already generated chromosomes and duplicates are discarded.

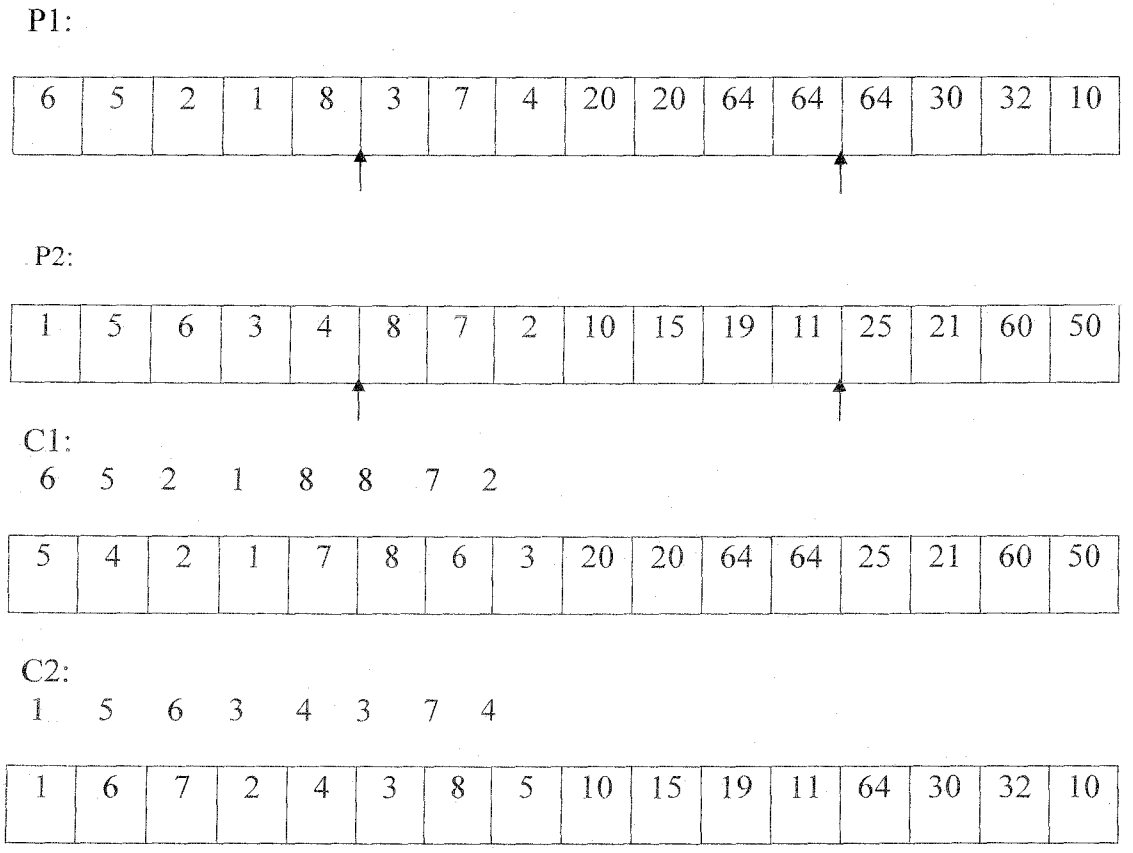


Figure 5.4: Example crossover operation

Figure 5.4 shows an example crossover operation. P1 and P2 are the parent chromosomes used to generate new children C1 and C2. C1 is generated by combining first part of P1 with second part of P2. Similarly, C2 is generated by joining the first part of P2 with the second part of P1.

While generating children in the above manner, we encounter the problem of two cores getting the same preference order. We resolve the problem in the following way. Let the order part of a chromosome with two cores having the same order be called *duplicate order part*. Our aim is to convert this *duplicate*

order part into *unique order part*. First of all, consider all cores with *preference* 1 from *duplicate order part*. The first such core is assigned *precedence* 1, while the second such core is allocated *preference* 2, and so on. Next, we consider all cores with *preference 2* from *duplicate order part* and assign next precedence to them. We continue this process until we get unique order part. For example, consider the *duplicate order part* of C2, mentioned in Figure 5.4. In the following, we show the steps to get the *unique order part* from this *duplicate order part*.

Duplicate order:	1	5	6	3	4	3	7	4
Step 1:	1	-	-	-	-	-	-	-
Step 2:	1	-	-	2	-	3	-	-
Step 3:	1	-	-	2	4	3	-	5
Step 4:	1	6	-	2	4	4	-	5
Step 5:	1	6	7	2	4	3	-	5
Step 6:	1	6	7	2	4	3	8	5

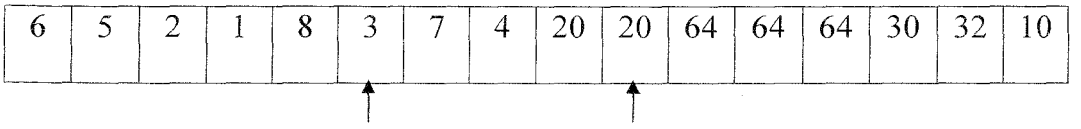
(Unique order part)

5.3.2.2 Mutation

Mutation is a very important operator as far as bringing variety into the population is concerned. As the population size is finite, the crossover operator alone cannot bring enough variation to the population. The mutation operator brings more effective variations into the chromosomes introducing newer search options. Figure 5.5 shows an example mutation operation. Here we take two separate mutation points for the ordering part and the rectangle-selection part. Based on the ordering mutation point, we change the preference of that particular core randomly. Similarly, based on the rectangle-selection mutation point, we choose a random width for the core selected. This gives another

solution for the problem. The duplicates in the ordering part are resolved similar to the crossover case.

P:



C:

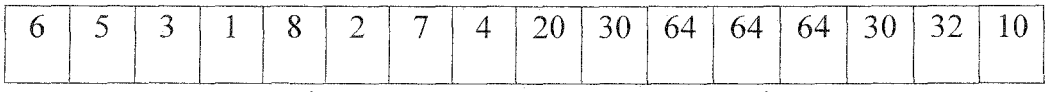


Figure 5.5: Example mutation operation

5.3.3 Fitness measure

Fitness of a chromosome is measured in terms of the cost of a solution, which is the total time required to test all cores in the system. The system testing time is the length of the rectangular bin after all cores are packed into that bin. The SOC test time is the fitness criteria in this method. The chromosome with minimum fitness value is the solution for the SOC.

5.3.4 Overall Genetic Algorithm

The evolution process is like this. We first generate the initial population chromosomes. The population size has been taken as 5000. Chromosomes are sorted in ascending order of their fitness values. First 20% chromosomes are copied directly to the next generation. We select 40% chromosomes by crossover and 40% chromosomes by mutation, as the population for the next generation. This process is repeated till no improvement in cost even after certain predefined number of generations (taken as 50 in our experimentation).

Algorithm GA

1. Create a random initial population of size POP_SIZE
2. For each chromosome *i* do

   2.1. Call *Place_rectangles(i)* to compute test time, T_i for chromosome *i*

   2.2. $Fitness_i = T_i$
3. Sort population
4. *No_of_gen_without_improv* = 0
5. While (*No_of_gen_without_improv* < 50) do

   5.1 Copy 20% best chromosomes to next generation

   5.2 Generate 30% new chromosomes via mutation

   5.3 Generate 50% new chromosomes via crossover

   5.4 Evaluate fitness of each chromosome using as in Step 2

   5.5 Sort population

   5.6 If (no improvement in fitness of best chromosome) then

   No_of_gen_without_improv ++

   Else

   No_of_gen_without_improv = 0

Procedure Place_rectangles(c)

/* **Possible_places:** an array of tuples <time_instant, wires_available> */

/* **No_places:** number of entries in array Possible_places */

1. No_places = 1
2. Possible_places[1] = <0, W> /* W is the total TAM wires available */
3. While all cores not placed do

3.1. $i = 1$

3.2. Let k be the next core chosen as per ordering part of c

3.3. Let the rectangle of k be R_k

3.4. Place R_k at time instant Possible_places[i].time_instant, if there is enough space to hold rectangle R_k

3.5. If R_k cannot be placed, choose a new core k from c; Goto Step 3.3

4. **Return** bin size as the total test time

End procedure

Algorithm 5.1: Overall Genetic Algorithm

5.3.5 Experimental results on unconstrained testing

In this section we present the experimental results. Table 5.3 presents the detailed comparison of our approach with other works reported in the literature. The approaches have been compared for 4 ITC'02 benchmark SOCs, namely d695 (10 cores), p93791 (32 cores), p34392 (21 cores) and p22810 (30 cores). For [86], we take the best of the two approaches reported there. To analyze the results, we have determined the best results in each case. The best results for each TAM width for a particular SOC has been shown in boldface in the table. Out of 28 such possible cases (7 for each SOC), [86] gives best results for 10 cases, [81] gives best results for 9 cases, our approach yields best results for 8 cases, whereas [87] gives the best result for a single case. Thus, our approach gives results comparable to the best ones known in the literature. It improves the basic scheme presented in [77] significantly. All the experiments have been carried out on a Pentium 4 machine with 1 MB main memory. The CPU time taken by the algorithm is a few seconds for the example SOCs. It may be noted that [181] presented another Genetic Algorithm based approach for SOC test scheduling. However, the results presented there are not on ITC'02

benchmarks. They have assumed some cores to have BIST within them. Thus, test times are not directly comparable with our work. Moreover, the total TAM width is also not mentioned for the examples that they have worked with.

5.4 Constraint driven scheduling

In this section, we extend the problem of integrated TAM scheduling to constraint driven test scheduling. The main constraints to be considered are

- Power constraints
- Precedence constraints

We solve the TAM scheduling problem such that the power and precedence constraints can be satisfied.

5.4.1 Power constraint

An efficient test schedule can reduce the testing time by allowing tests to be executed concurrently. However, executing tests concurrently increases the activity in the system, which leads to higher power consumption. It is important that the test power constraint must be considered carefully otherwise the system under test may be damaged due to overheating. SOCs in test mode can dissipate more power than in normal mode. This is because cores which do not normally operate in parallel may be tested concurrently to minimize testing time. Therefore, *power constrained test scheduling* is essential in order to limit the amount of concurrency during test application to ensure that the maximum power rating of the SOC is not exceeded. To take care of the power constraints, Step 3.4 of procedure **Place_rectangles** is modified to check the violation of power limit as well. The modified procedure looks as follows.

ITC'02 Benchmark	Strategy	Number of TAM wires						
		16	24	32	40	48	56	64
d965	Our	41891	28280	21307	17002	14420	12191	10744
	[86]	41553	**27982**	21014	**16908**	14236	**11988**	**10571**
	[87]	41847	29106	**20512**	18691	17257	-	13348
	[81]	**41604**	28064	21161	16993	**14183**	12085	10723
	[77]	43723	30317	23021	18459	15698	13415	11604
	[82]	41949	28327	21423	17210	16403	13023	12327
	[83]	44307	28576	21518	17617	14608	12462	11033
	[84]	46152	30777	22669	19551	16388	13877	11893
	[85]	42716	28639	21389	17366	15142	13208	11279
p93791	Our	**1743379**	1177300	886893	719191	**594168**	528255	447070
	[86]	1754980	1171190	88603	**706820**	600986	**501057**	**445748**
	[87]	1827819	1220469	945425	787588	639217	-	457862
	[81]	1757452	**1169945**	**878493**	718005	594575	509041	447971
	[77]	1851135	1248795	975016	794020	627934	568436	511286
	[82]	1775099	1192980	899807	705164	602613	521806	463707
	[83]	1791638	1185434	912233	718005	601450	528925	455738
	[84]	2404321	1598829	1179795	1060369	717602	625506	491496
	[85]	1791860	1200157	900798	719880	607955	521168	459233
p34392	Our	963000	657067	**544579**	**544579**	**544579**	**544579**	**544579**
	[86]	**939855**	637263	544579	544579	544579	544579	544579
	[87]	-	-	-	-	-	-	-
	[81]	944768	**628602**	544579	544579	544579	544579	544579
	[77]	1023820	759427	544579	544579	544579	544579	544579
	[82]	998733	720858	591027	544579	544579	544579	544579
	[83]	1010821	680411	551778	544579	544579	544579	544579
	[84]	1191829	828643	568133	544579	544579	544579	544579
	[85]	1016640	681745	544579	544579	544579	544579	544579
p22810	Our	**438306**	289780	226899	179897	151309	131180	117250
	[86]	438619	**289237**	226545	**167792**	153260	130949	116625
	[87]	473418	352834	236186	195733	159994	-	128332
	[81]	438619	289287	**218855**	175946	**147944**	**126947**	**109591**
	[77]	452639	307780	246150	197293	167256	145417	136941
	[82]	462240	361576	312662	278360	268474	266800	260639
	[83]	458068	299718	222471	190995	160221	145417	133405
	[84]	541402	356039	294788	220674	203257	175946	157527
	[85]	446684	300723	223462	184951	167858	145087	128512

Table 5.3: Test scheduling results.

Procedure Place_rectangles_power(c)

/* **Possible_places:** an array of tuples <time_instant, wires_available> */

/* **No_places:** number of entries in array Possible_places */

1. No_places = 1
   2. Possible_places[1] = <0, W> /* W is the total TAM wires available */
   3. While all cores not placed do
      3.1. $i = 1$
      3.2. Let k be the next core chosen as per ordering part of c
      3.3. Let the rectangle of k be R_k
      3.4. Place R_k at time instant Possible_places[i].time_instant, if there is enough space to hold rectangle R_k and power constraint is never violated throughout the schedule
      3.5. If R_k cannot be placed, choose a new core k from c; Goto Step 3.3
   4. **Return** bin size as the total test time

End procedure

5.4.1.1 Experimental results under power limitations

We applied our algorithm assuming different power constraint values. In the ITC'02 benchmark specification, no power data are given for benchmark files. We assumed the same power dissipation values for cores considered in the previous power constraint method [87]. Table 5.4 presents the results for SOC d695 for different power limit (P) values of 2500, 2000, 1800 and 1500. The total SOC testing time has been noted for different TAM widths. The results have been compared with those obtained in [87]. It can be seen that for SOC d695, the GA based approach needs, on an average, 9.34% lesser testing time than [87] under various power constraints. Table 5.5 presents the results for

SOC p22810 for different power limits and TAM widths. Here also the GA based approach needs 26.97% lesser testing time than [87]. Table 5.6 presents the results for SOC p93791 for various power limits and TAM widths. GA based approach needs 6.96% lesser testing time than [87] satisfying the power constraints.

Width	P=2500		P=2000		P=1800		P=1500	
	[87]	GA	[87]	GA	[87]	GA	[87]	GA
16	41847	42046	42450	41867	42450	42211	43541	42413
24	29106	28187	29106	28383	32054	28383	32663	29406
32	21931	21359	21942	21359	23864	21716	26973	22980
40	18691	16996	18691	17161	18774	17547	24369	18684
48	17257	14424	17467	15624	18774	16148	23425	17040
56	13963	12770	14563	12897	18774	13541	19402	15432
64	13394	12203	14469	12204	16804	12824	19402	15080

Table 5.4: Power constrained results for d695

Width	P=6000		P=5000		P=4000		P=3000	
	[87]	GA	[87]	GA	[87]	GA	[87]	GA
16	475961	441333	472026	438950	480223	441425	482963	443801
24	346461	292996	382507	293032	389243	296653	392525	303407
32	250487	226899	321930	229410	324478	226899	309255	226897
40	209559	189412	264038	189412	285307	189412	356215	192039
48	174928	156422	266166	156536	285814	156536	311632	155576
56	159686	145023	257600	135416	268272	145025	293528	147679
64	157568	124147	246110	116433	268856	121894	293012	121916

Table 5.5: Power constrained results for p22810

Width	P=2500		P=2000		P=1800		P=1500	
	[87]	GA	[87]	GA	[87]	GA	[87]	GA
16	1827819	1747112	1827819	1750012	1827819	1758396	1827819	1753643
24	1220469	1174193	1220469	1186000	1220469	11775316	1220469	1176422
32	965383	912246	957921	899338	1014616	904539	1117385	984578
40	821475	725531	821575	728513	848050	720951	1091210	870812
48	639217	603384	658132	606296	631214	599836	691866	602687
56	549669	532684	549669	525255	598487	528022	629051	571745
64	493599	445895	472653	458138	486469	469803	568734	508603

Table 5.6: Power constrained results for p93791

5.4.2 Precedence constraints

Precedence constraints impose a partial order among tests in a test suite. This can be motivated by several factors. For example, since BIST is likely to detect

more defects than an external test targeted only at random resistant faults, it may be desirable to apply BIST first to a core during manufacturing test. Similarly it may be desirable to test and diagnose memories earlier so that they can be used later for system test. Since larger cores are more likely to have defects due to their large silicon area, it may also be more desirable to test them first.

Let precedence constraints between cores be defined as, $i < j$ (test i must complete before test j is begun). We use the same problem formulation used in the generalized rectangle packing method but with the limitation that the precedence defined for that system must be satisfied. To take care of the precedence constraints, Steps 3.2 and 3.5 of procedure **Place_rectangles** are modified to check the violation of precedence. The procedure looks as follows.

Procedure Place_rectangles_precedence(c)

/* **Possible_places:** an array of tuples <time_instant, wires_available> */

/* **No_places:** number of entries in array Possible_places */

1. No_places = 1
2. Possible_places[1] = <0, W> /* W is the total TAM wires available */
3. While all cores not placed do
   - 3.1. $i = 1$
   - 3.2. Let k be the next core chosen as per ordering part of c satisfying precedence constrsints
   - 3.3. Let the rectangle of k be R_k
   - 3.4. Place R_k at time instant Possible_places[i].time_instant, if there is enough space to hold rectangle R_k
   - 3.5. If R_k cannot be placed, choose a new core k from c satisfying the precedence constraints; Goto Step 3.3

4. **Return** bin size as the total test time

End procedure

In the ITC'02 benchmark specification, no such precedence constraints are given for any system. We added some precedence constraints to SOC p22810 for simulation.

The added precedence constraints for SOC p22810 are:

- Core 12 can be tested after core 24
- Core 6 can be tested after core 12
- Cores 14 and 16 can be tested only after core 17
- Core 16 has to be tested before core 7
- Core 8 has to be tested after core 14
- Core 11 has to be tested after cores 7 and 8
- Cores 5 and 27 have to be tested after core 28
- Core 5 has to be tested before cores 1 and 2
- Core 27 has to be tested before cores 25 and 26
- Cores 1 and 25 have to be tested before core 20
- Cores 2 and 26 have to be tested before core 10
- Cores 10 and 20 have to be tested before core 15

Table 5.7 presents the testing time under the precedence constraints noted above for SOC p22810.

Width	Test Time
16	450562
24	304531
32	236521
40	193416
48	168569
56	161703
64	161588

Table 5.7: Results for SOC p22810 with given precedence constraints

5.5 Conclusion

In this chapter, we have presented a genetic algorithm based approach to solve the problem of System-on-Chip test scheduling using rectangular bin packing approach. For the unconstrained version of the problem, the scheme produces results comparable to the existing methods. The scheme has been extended to take care of the power constraints. The power constrained version produces results much improved than that reported in the literature. The basic scheme has also been modified for accommodating precedence constraints.